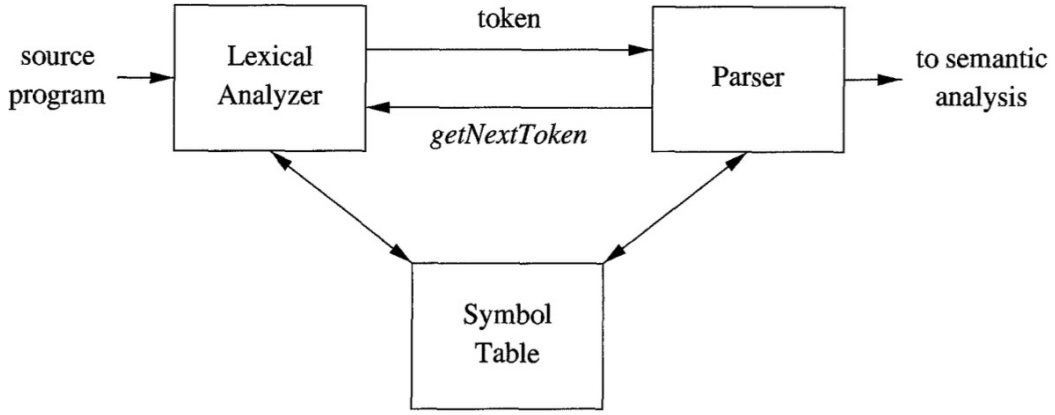
CS3025 Compiladores

Semana 3:
Analisis Sintactico
Gramaticas libres de Contexto / Context-free grammars
29 Agosto 2023

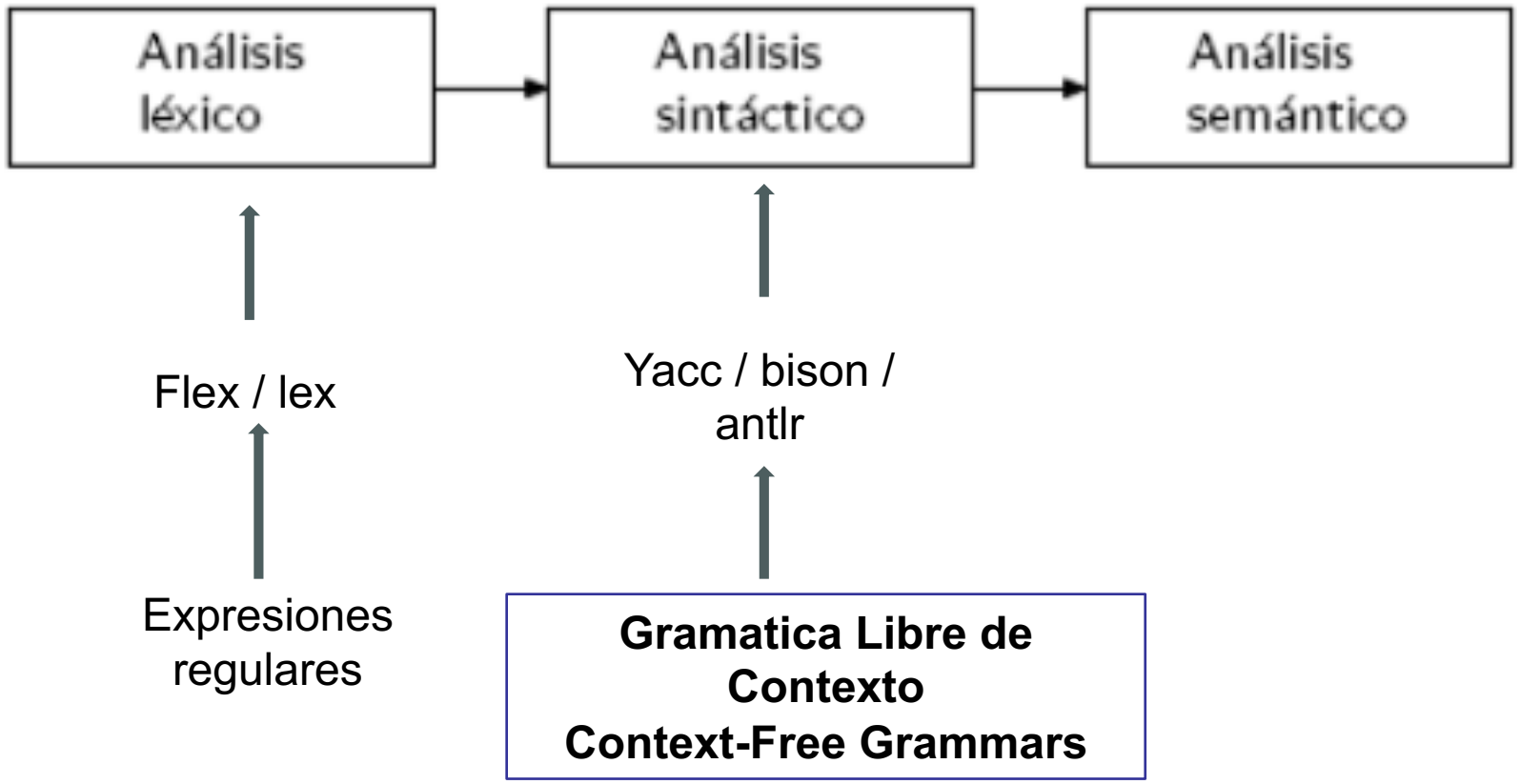
Igor Siveroni

Analisis Sintactico (Parser)



- El analizador sintáctico recibe el código fuente en la forma de tokens proveniente del analizador léxico.
- El análisis sintáctico determina los elementos estructurales del programa y sus relaciones.
- Los resultados del análisis sintáctico por lo regular se representan como un árbol de análisis gramatical o un árbol sintáctico.

Analisis Sintactico : Gramaticas libres de Contexto



Gramaticas Libres de Contexto / Context-Free Grammars

En el análisis léxico utilizamos el formalismo de expresiones regulares para definir el lenguaje de las palabras validas (unidades léxicas) de nuestro lenguaje (de programación).

Podemos usar el mismo formalismo para definir estructura en nuestros lenguajes?

Vimos que podemos usar el mecanismo de definiciones con expresiones regulares. Por ejemplo, las siguientes definiciones de *digits* y *sum*:

$$digits = [0-9]+$$
$$sum = (digits\ \text{“+”})^*\ digits$$

Define sumas de la forma 23 + 4 + 15. Ahora veamos las siguientes definiciones:

$$digits = [0-9]+$$
$$sum = expr\ \text{“+”}\ expr$$
$$expr = \text{“(”}\ sum\ \text{“)”} \mid digits$$

Gramaticas Libres de Contexto / Context-Free Grammars

La definición anterior pretende definir expresiones de la forma:

```
(109+23)
61
(1+(250+3))
```

en donde todos los paréntesis están balanceados. Pero sabemos que esto no lo puede hacer un automata finito: un automata de N estados no puede acordarse de mas de N parentesis por cerrar. La teoría nos dice que la definición anterior no es una expresión regular valida.

En realidad, la definición y uso de *digits* en la especificación inicial funciona como una macro, donde cada ocurrencia de *digits* se reemplaza por `[0-9]+`, lo cual es convertido a un autómata.

Abreviación se implementa vía sustitución.

Gramaticas Libres de Contexto / Context-Free Grammars

Si substituimos *sum* en *expr* en la definición anterior, obtenemos primero

$$expr = \text{“ (”}\ expr\ \text{“+”}\ expr\ \text{“) ”} \mid digits$$

y luego substituimos *expr* (izquierda) y *sum* una vez obtenemos

$$expr = \text{“ (”}\ (\ \text{“ (”}\ expr\ \text{“+”}\ expr\ \text{“) ”} \mid digits\ )\ \text{“+”}\ expr\ \text{“) ”} \mid digits$$

Donde simplemente continuamos expandiendo la formula – esto no nos ayuda.

El uso de definiciones recursivas es poderoso pero combinado con definiciones interpretadas como substituciones no nos da mas poder.

Buscamos otro paradigma: el de producciones y derivaciones
¡Gramáticas Libres de contexto!

Gramaticas Libres de Contexto / Context-Free Grammars

Seguimos considerando a un lenguaje como un conjunto de cadenas, donde cada cadena es una secuencia finita de símbolos tomados de un alfabeto finito.
Para el análisis sintáctico, las cadenas son los programas fuente, los símbolos tokens léxicos, y el alfabeto el conjunto de tipos de token retornados por el analizador léxico (scanner).

Las gramáticas libres de contexto / context-free grammars (Gramáticas de ahora en adelante) describen lenguajes. Una gramática es un conjunto de *producciones* de la forma:

$$symbol \rightarrow symbol\ symbol \cdots symbol$$

con cero o mas símbolos en la parte derecha de la producción. Cada símbolo es:

- Un terminal: un token del alfabeto del lenguaje, o
- Un no-terminal: un símbolo que que aparece al lado izquierdo de otra producción.

Además,

- Ningún token puede aparecer al lado izquierdo de una producción, y
- Uno de los no-terminales es señalado como símbolo inicial de la gramática. Si no es señalado explícitamente, se asume que es el de la primera producción.

Expresiones Regulares subconjunto de Gramaticas Libres de Contexto

Notese que las gramáticas pueden usarse para describir expresiones regulares, es decir, el conjunto de lenguajes definidos por expresiones regulares es un subconjunto del conjunto del lenguajes definidos por grmaticas.

Por ejemplo:

$$expr = ab(c \mid d)e$$ es equivalente a

$$\begin{aligned} aux &= c \\ aux &= d \\ expr &= a\ b\ aux\ e \end{aligned}$$

De igual modo

$$expr = (a\ b\ c)*$$ es equivalente a

$$\begin{aligned} expr &= (a\ b\ c)\ expr \\ expr &= \epsilon \end{aligned}$$

Ejemplo: un lenguaje imperativo

1 $S \rightarrow S\ ;\ S$
2 $S \rightarrow \text{id} := E$
3 $S \rightarrow \text{print}\ (\ L\ )$
4 $E \rightarrow \text{id}$
5 $E \rightarrow \text{num}$
6 $E \rightarrow E + E$
7 $E \rightarrow (S\ ,\ E\ )$
8 $L \rightarrow E$
9 $L \rightarrow L\ ,\ E$

Consideremos la sintaxis de un lenguaje de programación imperativo especificada por la gramática de arriba:

- El símbolo inicial o de partida es S (dado que no ha sido mecionado explícitamente).
- Los símbolos terminales son: `id print num , + ( ) := ;`
- Lon símbolos terminales son *S, E y L*.

Una oracion del lenguaje definido por esta gramática es

```
id := num; id := id + (id := num + num, id)
```

Donde el código fuente pudo haber sido

```
a := 7;
b := c + (d := 5 + 6, d)
```

Los tipos de token son `id, num, :=,` etc, y los nombres (a,b,c,d) y los números (7,5,6) son los *valores semánticos* asociados a algunos de los tokens.

Ejemplo: derivaciones

Para demostrar que una oración pertenece al lenguaje de una gramática, debemos efectuar una derivación: empezando por el símbolo de inicio, reemplazar repetidamente cualquier no-terminal por cualquiera de sus expansiones (mano derecha en la gramática).

A la derecha, una derivación de la gramática en cuestión.

Podemos verificar que la derivación se produce tras aplicar sucesivamente las reglas 1, 2, 2, 5, 6, etc a los no-terminales subrayados.

Nótese que la elección del no-terminal a expandir no tiene ningún patron.

$$\underline{S}$$
$$S\ ;\ \underline{S}$$
$$\underline{S}\ ;\ \mathrm{id} := E$$
$$\mathrm{id} := \underline{E}\ ;\ \mathrm{id} := E$$
$$\mathrm{id} := \mathrm{num}\ ;\ \mathrm{id} := \underline{E}$$
$$\mathrm{id} := \mathrm{num}\ ;\ \mathrm{id} := E + \underline{E}$$
$$\mathrm{id} := \mathrm{num}\ ;\ \mathrm{id} := \underline{E} + (S\ ,\ E)$$
$$\mathrm{id} := \mathrm{num}\ ;\ \mathrm{id} := \mathrm{id} + (\underline{S}\ ,\ E)$$
$$\mathrm{id} := \mathrm{num}\ ;\ \mathrm{id} := \mathrm{id} + (\mathrm{id} := \underline{E}\ ,\ E)$$
$$\mathrm{id} := \mathrm{num}\ ;\ \mathrm{id} := \mathrm{id} + (\mathrm{id} := E + E\ ,\ \underline{E})$$
$$\mathrm{id} := \mathrm{num}\ ;\ \mathrm{id} := \mathrm{id} + (\mathrm{id} := \underline{E} + E\ ,\ \mathrm{id})$$
$$\mathrm{id} := \mathrm{num}\ ;\ \mathrm{id} := \mathrm{id} + (\mathrm{id} := \mathrm{num} + \underline{E}\ ,\ \mathrm{id})$$
$$\mathrm{id} := \mathrm{num}\ ;\ \mathrm{id} := \mathrm{id} + (\mathrm{id} := \mathrm{num} + \mathrm{num}\ ,\ \mathrm{id})$$

Derivaciones por la izquierda y por la derecha

Puede haber mas de una derivación para una oración.
Una derivación por la izquierda (leftmost derivation) es la que expende siempre el no-terminal ubicado mas a la izquierdo. En una derivación por la derecha (rightmost derivation), es el no-terminal mas a la derecha el que se expande siempre.

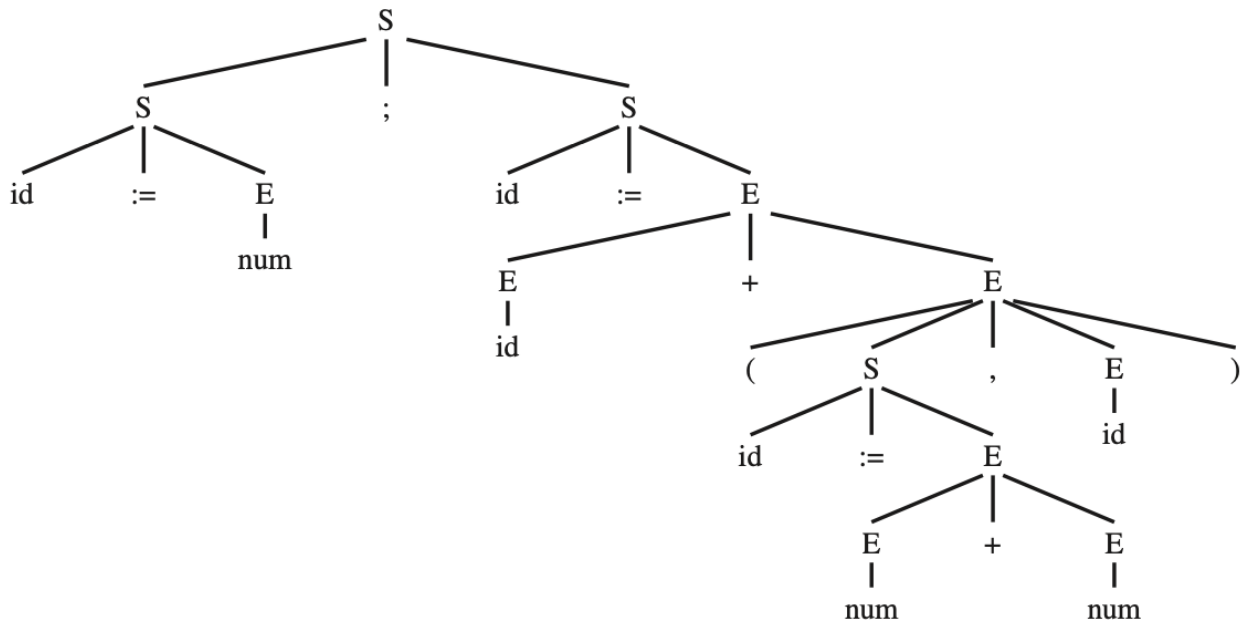
La derivación de la pagina anterior no pertenece a estas categorías. Abajo mostramos una derivación por la izquierda:

$$
\begin{aligned}
&\underline{S}\\
&\underline{S}\ ;\ S\\
&\mathrm{id}\ :=\ \underline{E}\ ;\ S\\
&\mathrm{id}\ :=\ \mathrm{num}\ ;\ \underline{S}\\
&\mathrm{id}\ :=\ \mathrm{num}\ ;\ \mathrm{id}\ :=\ \underline{E}\\
&\mathrm{id}\ :=\ \mathrm{num}\ ;\ \mathrm{id}\ :=\ \underline{E}\ +\ E\\
&\quad\vdots
\end{aligned}
$$

Arboles de Sintaxis (Parse Trees)

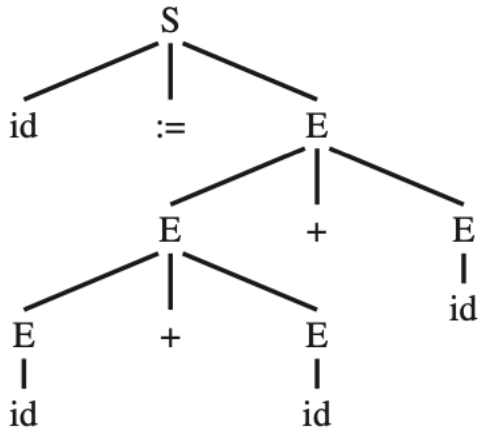
Un árbol de sintaxis se genera al conectar los símbolos de una derivación con los símbolos de la derivación de la que fue generada. Dos derivaciones distintas pueden terner el mismo árbol de sintaxis.
Por ejemplo, el siguiente árbol es generado de la derivacion de la izquierda:

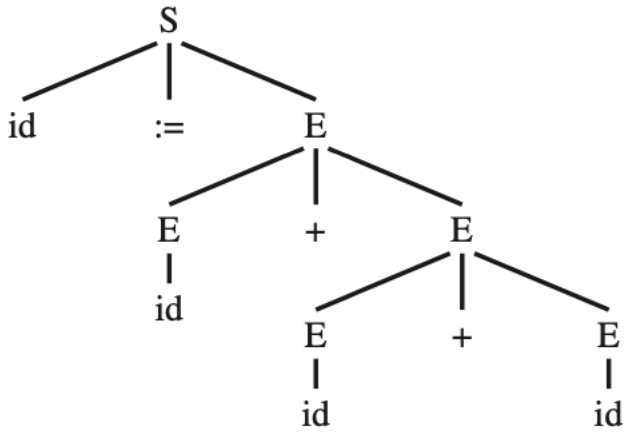
$\underline{S}$
$S\ ;\ \underline{S}$
$\underline{S}\ ;\ \text{id} := E$
$\text{id} := \underline{E}\ ;\ \text{id} := E$
$\text{id} := \text{num}\ ;\ \text{id} := \underline{E}$
$\text{id} := \text{num}\ ;\ \text{id} := E\ +\ \underline{E}$
$\text{id} := \text{num}\ ;\ \text{id} := \underline{E}\ +\ (S\ ,\ E\ )$
$\text{id} := \text{num}\ ;\ \text{id} := \text{id}\ +\ (\underline{S}\ ,\ E\ )$
$\text{id} := \text{num}\ ;\ \text{id} := \text{id}\ +\ (\text{id} := \underline{E}\ ,\ E\ )$
$\text{id} := \text{num}\ ;\ \text{id} := \text{id}\ +\ (\text{id} := \underline{E}\ +\ E\ ,\ \underline{E}\ )$
$\text{id} := \text{num}\ ;\ \text{id} := \text{id}\ +\ (\text{id} := \underline{E}\ +\ E\ ,\ \text{id}\ )$
$\text{id} := \text{num}\ ;\ \text{id} := \text{id}\ +\ (\text{id} := \text{num}\ +\ \underline{E}\ ,\ \text{id}\ )$
$\text{id} := \text{num}\ ;\ \text{id} := \text{id}\ +\ (\text{id} := \text{num}\ +\ \text{num}\ ,\ \text{id}\ )$



Gramaticas Ambiguas I

Una gramática es ambigua se puede derivar una oración con dos arboles de sintaxis diferentes.
Por ejemplo, la oración `id := id+id+id` tiene 2 arboles de sintaxis:





Vemos que ,la suma asocia a la izquierda, en un caso, y a la derecha en otro.

Gramaticas Ambiguas II

$E \rightarrow \text{id}$

$E \rightarrow \text{num}$

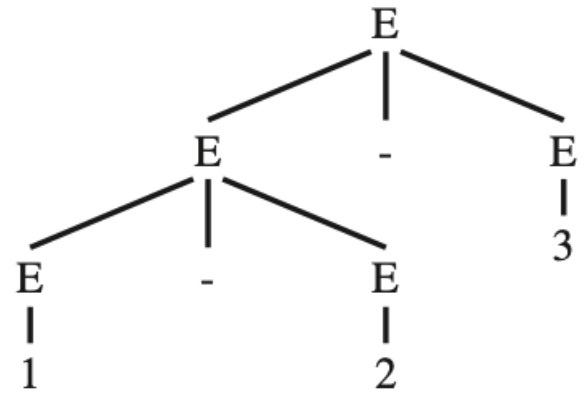
$E \rightarrow E * E$

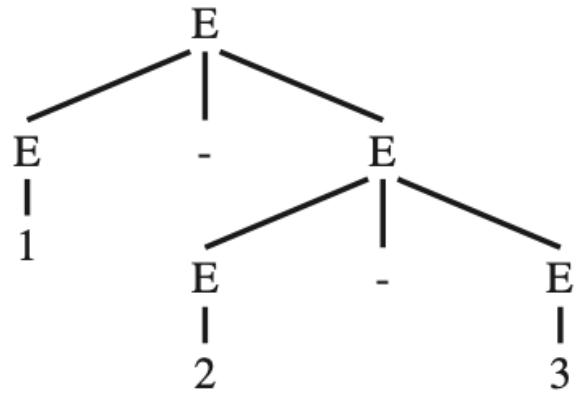
$E \rightarrow E / E$

$E \rightarrow E + E$

$E \rightarrow E - E$

$E \rightarrow (E)$





Oracion: `1-2-3`

Gramaticas Ambiguas III

$E \rightarrow \text{id}$

$E \rightarrow \text{num}$

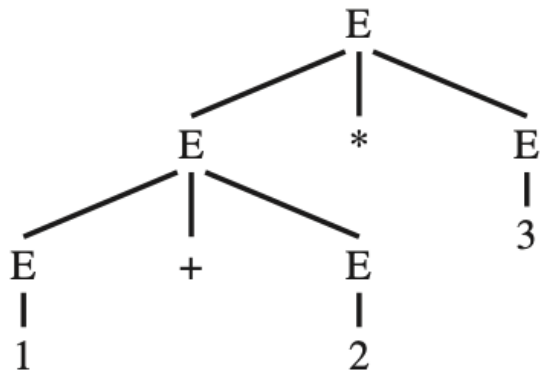
$E \rightarrow E * E$

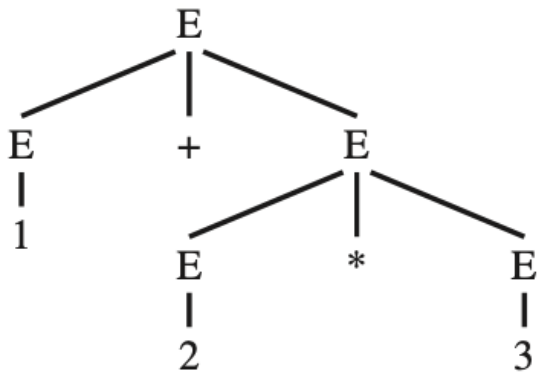
$E \rightarrow E / E$

$E \rightarrow E + E$

$E \rightarrow E - E$

$E \rightarrow (E)$





Oracion: `1+2*3`

Gramaticas Ambiguas: Significado es importante

En los dos últimos casos, la interpretación del significado de las expresiones aritméticas denotadas por los arboles de sintaxis nos da resultados diferentes!
Esto es problemático: compiladores usan arboles de sintaxis para derivar significado.

Las gramáticas ambiguas no son amigas de los compiladores.
Es preferible trabajar con gramáticas no ambiguas. Afortunadamente, generalmente es posible transformar gramáticas ambiguas o gramáticas que no lo son.

Encontremos una gramática no ambigua equivalente a la gramática de expresiones aritméticas presentada en las paginas anteriores.
La nueva gramática deberá seguir lo estándar al evaluar expresiones aritméticas:

- Multiplicación y división tienen mayor precedencia que suma y resta.
- Los operadores asocian “a la izquierda”, es decir, al derivar `1-2-3` obtenemos `(1-2)-3` en lugar de `1-(2-3)`.

Nueva gramatica para aexp

$$E \to E + T \qquad T \to T * F \qquad F \to \text{id}$$
$$E \to E - T \qquad T \to T / F \qquad F \to \text{num}$$
$$E \to T \qquad T \to F \qquad F \to (E)$$

La nueva gramática NO es ambigua.

Los símbolos E, T y F denotan expresión, termino y factor.
Factores son elementos que se multiplican entre si, mientras que los términos son los que se suman entre si.

Esta gramática acepta el mismo conjunto de oraciones que la gramática ambigua anterior.

Simbolo (token) de fin de archive (EOF)

$$S \rightarrow E\ \$$$

$$E \rightarrow E + T$$
$$E \rightarrow E - T$$
$$E \rightarrow T$$

$$T \rightarrow T * F$$
$$T \rightarrow T / F$$
$$T \rightarrow F$$

$$F \rightarrow \text{id}$$
$$F \rightarrow \text{num}$$
$$F \rightarrow (E)$$

Los analizadores gramaticales no solo leen símbolos terminales con +, -,(,), etc, también deben de leer un símbolo especial que denote fin de input, o fin de archivo.

La nueva gramática de arriba ha sido extendida para hacer explicita la lectura del EOF, usando el símbolo $.

Nuestros scanners y parsers usan un token especial: EOF (end-of-file).

Gramaticas y analisis sintactico

El proceso de análisis sintáctico puede verse como el proceso de buscar una derivación dada la cadena de entrada.

Entonces, el diseño de la gramática es importante.

Existen ciertas gramáticas que son fáciles de analizar utilizando el algoritmo de **descenso recursivo o análisis predictivo**.
El algoritmo :

- Tiene una función por cada no-terminal
- Y una clausula por cada produccion

- Para esto es necesario que el primer símbolo terminal de cada producción provea suficiente información para seleccionar la producción a utilizar

Ejemplo: Descenso recursivo

$S \rightarrow \text{if } E \text{ then } S \text{ else } S$
$S \rightarrow \text{begin } S\ L$
$S \rightarrow \text{print } E$

$L \rightarrow \text{end}$
$L \rightarrow ;\ S\ L$

$E \rightarrow \text{num} = \text{num}$

```
final int IF=1, THEN=2, ELSE=3, BEGIN=4, END=5
          SEMI=7, NUM=8, EQ=9;

int tok = getToken();

void advance() {tok=getToken();}
void eat(int t) {if (tok==t) advance(); else e

void S() {switch(tok) {
        case IF:    eat(IF); E(); eat(THEN); S
                    eat(ELSE); S(); break;
        case BEGIN: eat(BEGIN); S(); L(); brea
        case PRINT: eat(PRINT); E(); break;
        default:    error();
       }}
void L() {switch(tok) {
        case END:   eat(END); break;
        case SEMI:  eat(SEMI); S(); L(); break
        default:    error();
       }}
void E() {  eat(NUM); eat(EQ); eat(NUM); }
```

Descenso recursivo: funciona con la gramatica de expresiones aritmeticas?

$$S \rightarrow E\ \$$$

$$E \rightarrow E + T \qquad E \rightarrow E - T \qquad E \rightarrow T$$

$$T \rightarrow T * F \qquad T \rightarrow T / F \qquad T \rightarrow F$$

$$F \rightarrow \text{id} \qquad F \rightarrow \text{num} \qquad F \rightarrow (\ E\ )$$

```
void S() {  E(); eat(EOF); }
void E() {switch (tok) {
          case ?: E(); eat(PLUS); T(); break;
          case ?: E(); eat(MINUS); T(); break;
          case ?: T(); break;
          default: error();
         }}
void T() {switch (tok) {
          case ?: T(); eat(TIMES); F(); break;
          case ?: T(); eat(DIV); F(); break;
          case ?: F(); break;
          default: error();
         }}
```

El método de descenso recursivo, aplicado a la gramática se vería como el código a la izquierda pero … el método no sabe que regla aplicar dado un token!

Por ejemplo, la expresión (1*2 -3)+4 debería usar la produccion

$$E \rightarrow E + T$$

Mientras que la expresión (1*2-3) debería usar:

$$E \rightarrow T$$

Que paso? La gramática es left-recursive

Descenso Recursivo (Parseo Predictivo) y conjuntos FIRST(A)

El algoritmo de descenso recursivo:

- Tiene una función por cada no-terminal.
- Las funciones tienen una clausula por cada producción / regla.
- Para esto es necesario que, dado un token, sea posible seleccionar la producción a expandir.

La tercera condición se puede generalizar definiendo el conjunto FIRST(A) de todos los símbolos terminales que pueden empezar la producción A, e.g.

$$\text{FIRST}(T * F) = \{\texttt{id}, \texttt{num}, \texttt{(}\}$$

Un algoritmo de descenso recursivo utiliza el conjunto FIRST(A) para crear una tabla que guía la elección de producciones a expandir (explorar).
Consideremos 2 producciones diferentes, A -> A1 y X -> A2:

- Si la intersección de los conjuntos FIRST(A1) y FIRST(A2) es no vacía, es decir, existe un símbolo terminal que aparece en ambos conjuntos, la gramática no puede ser analizada usando descenso recursivo.
  El algoritmo no tiene manera de decidir cual producción usar!!

Descenso Recursivo (Parseo Predictivo) y gramaticas LL(1)

La clase de gramáticas que pueden ser analizadas usando descenso recursivo, aquellas cuyas tablas de análisis predictivo no contienen duplicados, se llama **LL(1)**

LL(1)

left-to-right parse, leftmost-derivation, 1-symbol lookahead

- *Left-to-right*: el input es examinado de izquierda a derecha en una sola pasada.
- *Leftmost-derivation*: El orden en el que un analizador predictivo expande no-terminales corresponde a una derivación por la izquierda.
- Un analizador de descenso recursivo solo requiere información del siguiente token. A esto se le llama 1 lookahead.
  Esto ultimo se puede generalizar a *k* tokens: gramáticas LL(k)

Que tipo de gramáticas son problemáticas?

- Ambiguas
- Gramáticas con recursión a la izquierda (*left recursion*)
- Gramáticas que requieren factorización a la izquierda (*left factoring*)

Gramaticas LL(1) y recursion a la izquierda

Considérese la gramática de expresiones aritméticas. Las dos producciones de la izquierda generarían duplicados en la tabla de análisis predictivo:

$$E \rightarrow E + T$$
$$E \rightarrow T$$

$$E \rightarrow T\ E'$$
$$E' \rightarrow +\ T\ E'$$
$$E' \rightarrow$$

dado que todo token en FIRST(T) será parte de FIRST(E + T). El problema es causado por la presencia de E como primer símbolo en una de las producción de E. Esto llama *left recursion* (recursión a la izquierda). Gramáticas con *left recursion* no pueden ser LL(1).

Es posible eliminar la recursión a la izquierda reescribiendo la producción con recursión a la derecha. El resultado de la transformación se muestra a la derecha de la producción original (arriba).

Eliminando recursion a la izquierda

Si aplicamos la transformación que elimina `left recursión` a la gramática de expresiones aritméticas, obtenemos lo siguiente

$$S \rightarrow E\ \$$$

$$E \rightarrow T\ E'$$

$$E' \rightarrow +\ T\ E'$$
$$E' \rightarrow -\ T\ E'$$
$$E' \rightarrow$$

$$T \rightarrow F\ T'$$

$$T' \rightarrow *\ F\ T'$$
$$T' \rightarrow /\ F\ T'$$
$$T' \rightarrow$$

$$F \rightarrow \text{id}$$
$$F \rightarrow \text{num}$$
$$F \rightarrow (\ E\ )$$

La gramática es ahora LL(1) – podemos escribir un analizador usando descenso recursivo! (usando conjuntos FIRST(A))

Pero hay otra opción

Eliminando recursion a la izquierda

Volvamos a la parte de la gramática que define expresiones:

$$E \rightarrow E + T$$
$$E \rightarrow E - T$$
$$E \rightarrow T$$

Esta parte de la gramática en realidad este definiendo E -> T ((´+´|´-´) T)*
Es decir, una secuencia de Términos T intercalados con operadores ´+´ y ´–´

Esto puede traducirse fácilmente en código siguiendo el método de descenso recursivo:

```
void Parser::parseExpression() {
  parseTerm();
  while(match(Token::MINUS) || match(Token::PLUS)) {
    parseTerm();
  }
  return;
}
```

Eliminando recursion a la izquierda

El resto del código (ver parser_aexp.cpp) se escribiría asi:

```
void Parser::parse() {
  current = scanner->nextToken();
  parseExpression();
  if (current->type != Token::END) {
    cout << "Esperaba fin-de-input”;
    … error
  }
  return;
}
```

```
// F ((´*´|´/´) F)*
void Parser::parseTerm() {
  parseFactor();
  while(match(Token::MULT)||
        match(Token::DIV))
    parseFactor();
  }
  return;
}
```

Eliminando recursion a la izquierda

El resto del código (ver parser_aexp.cpp) se escribiría asi:

$$F \rightarrow \text{id}$$
$$F \rightarrow \text{num}$$
$$F \rightarrow (E)$$

```
void Parser::parseFactor() {
  if (match(Token::NUM)) return;
  if (match(Token::LPAREN)) {
    parseExpression();
    if (!match(Token::RPAREN)) {
      cout << "Expecting right parenthesis" << endl;
      exit(0);
    }
    return;
  }
  cout << "Couldn't find match for token: " << current->lexema << endl;
  exit(0);
}
```